OneMax

El problema de OneMax (o BitCounting) es un problema que consiste en generar una cadena de bits que contenga la mayor cantidad de Unos (1)s.

El problema en sí es muy simple y ampliamente utilizado en la educación de la comunidad computacional evolutiva.

Formalmente, este problema se puede describir como encontrar una cadena $x = \{x_1, x_2, \ldots, x_N\}$, con $x_i \in \{0, 1\}$, que maximize la siguiente función:

$$F(x) = \sum_{i=1}^{N} x_i$$

Esta será nuestra *función de aptitud* (función de evaluación).

Crearemos una población de individuos compuesta de vectores enteros llenos aleatoriamente con 0 y 1.

Luego nuestra población evolucione hasta que uno de sus miembros contenga solo 1 y no 0 más.

Herramientas

- Escogimos el lenguaje de programación *Python*, por ser muy eficiente en cuando al analisis de datos y la ciencia de datos.
- Utilizamos DEAP, que es un framework especializado en la computación evolutiva.
- La interfaz estará a cargo de una aplicación web llamada *Jupyter*

Descripción del algoritmo

Importación de Librerias

In [ ]:
```
import array
import random
import numpy
from deap import algorithms, base, creator, tools
```

Nuestra función de evaluación:

In [ ]:
```
def func_eval(individuo):
    # Sumatoria de 1s en el individuo.
    return sum(individuo),
```

Clase que hereda de Fitness (Mide la calidad de una solución, y le envian los valores iniciales en tupla(1.0))

In [ ]:
```
creator.create("AptitudMax", base.Fitness, weights=(1.0,))
# Crea otra clase de con un array de tipo binario.
creator.create("Individuo", array.array, typecode='b', fitness=creator.AptitudMax)
```

Inicializa la "caja" de herramientas, para configurarla a nuestro gusto.

In [ ]:
```
toolbox = base.Toolbox()
```

Generador de los atributos (0 y 1)

In [ ]:
```
toolbox.register("attr_bool", random.randint, 0, 1)
```

Inicializadores

In [ ]:
```
# En este caso queremos que el mejor individuo contenga 20 uno (1)s
toolbox.register("individuo", tools.initRepeat, creator.Individuo, toolbox.attr_bool, 20)
# La población se rellena de listas de individuos
toolbox.register("poblacion", tools.initRepeat, list, toolbox.individuo)

# Registra el operador de evaluación (nuestra función)
toolbox.register("evaluate", func_eval)

# Registra el operador de cruze (dos puntos)
toolbox.register("mate", tools.cxTwoPoint)

# Resgistra el operador de muación, con la probabilidad del 5% que cada uno sea convertido
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)

# Operador para seleccionar los individuos que servirán para reproducción,
# tournsize es el numero de individuos participando
toolbox.register("select", tools.selTournament, tournsize=3)
```

Aquí definimos todas la lógica que va desarrollar el algoritmo.

Los parametros iniciales fueron:

- Población: 300
- Número de generaciones: 5
- Probabilidad de Mutación: 50%
- Probabilidad de Cruce: 20%

Semilla para la aleatoriedad, es necesaria ya que los numeros generados por un lenguaje de programación son pseudoaleatorios, y necesitan una semilla para comenzar a generar, si no se indica, generalmente obtienen la hora del ordenador.

In [ ]:
```
random.seed(64)
```

Se indica el tamaño de la población

In [ ]:
```
poblacion = toolbox.poblacion(n=300)
```

Contiene el mejor individuo que haya vivido en la población durante la evolución.

In [ ]:
```
hof = tools.HallOfFame(1)
```

Aquí definimos las estadisticas que queremos que sean tomadas en cuenta ya arrojadas en el log del programa. Nos interesa el mínimo, el máximo y el promedio de 1s en el individuo.

In [ ]:
```
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("Promedio", numpy.mean)
stats.register("min", numpy.min)
stats.register("max", numpy.max)
```

DEAP, nos facilita todo el ciclo de evaluar cada inidviduo de la población y nos entrega un algoritmo que lo realiza de una manera más eficiente.

Este algoritmo por defecto que toma los valores de probabilidad de cruze (cxpb), Probabilidad de mutación (mutpb), Numero de Generaciones (ngen), las stadisticas que queremos mostrar (stats) Retorna una tupla con el log, y la población final.

In [ ]:
```
poblacion, log = algorithms.eaSimple(poblacion, toolbox, cxpb=0.5, mutpb=0.2, ngen=5,
                                     stats=stats, halloffame=hof, verbose=True)
```

Una vez que tenemos el log y la población final (luego de las generaciones), podemos seleccionar un top de los mejores individuos, o simplemente el mejor.

Mejor Individuo:

```
array('b', [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

Número de Unos: 20